VTT Technical Research Centre of Finland

Model checking as a protective method against spurious actuation of industrial control systems

Pakonen, Antti; Björkman, Kim

Published in:
Safety and Reliability

DOI:
10.1201/9781315210469

Published: 01/01/2017

Document Version
Peer reviewed version

Model checking as a protective method against spurious actuation of industrial control systems

A. Pakonen & K. Björkman
VTT Technical Research Centre of Finland Ltd, Espoo, Finland

ABSTRACT: A spurious actuation of an industrial instrumentation and control (I&C) system is a failure mode where a system component inadvertently performs a function without a justified reason to do so. Design issues leading to such failures are very difficult to analyze, but pose a high risk for safety. Model checking is a formal verification method, that can enable – through exhaustive analysis against stated properties – 100% coverage against spurious actuation scenarios, as well. In this paper, we introduce a modeling approach for the verification of I&C system application logic design. We then discuss the verification of properties related to spurious actuation, in particular. Finally, we present data collected from customer projects VTT has carried out in the Finnish nuclear industry. About 37% of the design issues we have identified are related to spurious actuation, proving that model checking can effectively be used as a protective method against such scenarios.

1 INTRODUCTION

"If the pressure gets too high, an emergency relief valve shall be automatically opened." Given such a requirement, a test engineer should have little trouble coming up with a suitable verification plan. But actually, a more critical statement might be the counterpoint: "The emergency relief valve shall *never* be opened unless it is *absolutely necessary*". Given such a requirement to verify, where does one even start?

Spurious actuation is defined as a failure mode where an actuation of an I&C function occurs without a real demand (Authén et al. 2016). The terms "inadvertent operation" or "active failure" are also used. By their nature, such failures are more complex to analyze than "failure to actuate". Spurious actuation can be caused by any failure between the process measurement sensors and the actuators, including erroneous operator command (Authén et al. 2016). In this paper, however, we focus on failures caused by design issues in the I&C system application logic.

One of the safety design principles in nuclear power plants (NPPs) is defense in depth – establishment of several successive physical barriers for containing accidents. Still, spurious actuation of a safety I&C system is a hazard that can potentially challenge more than one barrier simultaneously (IAEA 2015).

The common position of several nuclear regulators (2015) is that safety of systems "cannot be discussed and shown to exist" without accurate descriptions of system architecture, hardware, software, models of postulated accidents, etc., and that those descriptions "must include unintended systems behaviour". Several regulators also agree that spurious actuation is of "particular concern from a safety standpoint", and "has the potential to place a given plant into an unsafe operation condition that is not bounded by the plant's safety analysis" (MDEP 2016).

One NPP design principle that can be used to deal with spurious actuation is diversity – adding another, different I&C system, and then voting on control actions. Still, aside from emphasizing good principles like defense in depth, single failure tolerance, quality, independence and qualification (IAEA 2015), there remains a challenge: How the ensure that I&C systems do not contain design errors that might lead to spurious actuation? Modern I&C systems are so complex, in terms of both hardware and software (platform and application), that 100% test coverage is practically impossible.

The contribution of this paper is threefold. First, after discussing basic aspects of model checking in Section 2, we introduce a modeling approach developed for the verification of I&C system application logic in Section 3. Second, we discuss the semantics of and relationships between different types of formal properties that deal with spurious actuation in Section 4. Third, in Section 5, we present data on design issues identified in VTT's practical customer projects, to support our argument that model checking is a powerful method for analyzing spurious actuation, in particular. Finally, our conclusions are stated in Section 6.

2 FORMAL VERIFICATION

2.1 *Model checking*

Model checking is a formal verification method by which a desired property of a (hardware or software) system is verified over a system model through exhaustive enumeration of all the reachable states and possible behaviors (Clarke et al. 1999). When the design fails to satisfy a desired property, the *model checker* (a software tool used for analysis) produces a *counterexample* that demonstrates a behavior that violates the property.

The key challenge in model checking is the *state space explosion* problem. For systems with many interacting components or data structures that can assume many different values, the number of possible states to enumerate through becomes too enormous (Clarke et al. 1999). One solution is to use symbolic representation of the state space. Binary Decision Diagrams (BDD) provide a canonical representation for boolean formulas, and allow for verification of systems that would be impossible to handle using explicit state enumeration (Burch et al. 1992). Another solution is to use Boolean (or propositional) satisfiability (SAT) solvers to perform *bounded model checking*, where the length of state transition sequences is limited (Clarke et al. 2001).

NuSMV (Cimatti et al. 2002), a popular open source model checker, is one example of a BDD and SAT-based verification tool. The system model is written as a type of Finite State Machine (FSM). NuSMV is based on discrete time, but continuous time model checkers are also available (e.g., UPPAAL (Behrmann et al. 2004)).

2.2 *Property specification*

Formal properties are often divided into different types: *safety* properties dictate that something "bad" will not happen, and *liveness* dictate that something "good" must eventually happen (Lamport 1977). More accurately, we can distinguish between these two types by the fact that a *finite* execution of a system cannot violate a liveness property (since it is always possible that the "good" thing might occur later). On the other hand, an execution that violates a safety property is always limited, and there is an identifiable point where the "bad" thing occurs (Alpern and Schneider 1985). The kind of properties that deal with spurious actuation are therefore safety properties.

For model checking, the properties are usually specified using *temporal logic*, a formalism that describes sequences of transitions between states in a system (Clarke et al. 1999). Languages such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) utilize *temporal operators*, such as (using notation from (Clarke et al. 1999)):

- $\mathbf{X}\ p : p$ is true in the next state of the path.
- $\mathbf{G}\ p : p$ is true at every state on the path.
- $\mathbf{F}\ p : p$ is true at some future state on the path.
- $p\ \mathbf{U}\ q : q$ is true at some future state, and at every preceding state on the path, p is true.

In these expressions, p and q are *atomic propositions*, logical formulas (referring to system variables) that will either be true or false in any system state.

It is often also convenient to be able to refer to past states. Past LTL operators have been suggested by, e.g., Benedetti and Cimatti (2003), and the following operators are also among those supported by tools such as NuSMV:

- $\mathbf{Y}\ p : p$ holds in the previous state on the path.
- $\mathbf{O}\ p : p$ is true at some past (or the current) state on the path.

CTL is based on branching time, adding path quantifiers **A** ("for all computation paths") and **E** ("for some computation path").

3 VERIFICATION OF I&C SYSTEMS IN PRACTICE

3.1 *Practical experience*

Since 2008, VTT has applied model checking in practical customer projects in the Finnish nuclear industry. Our clients include (but are not limited to) the nuclear regulator and two utilities.

For the Finnish Radiation and Nuclear Safety Authority (STUK), we have evaluated the Protection System and the Priority and Actuation Control System of the Olkiluoto 3 NPP under construction. Both systems are based on Areva's TELEPERM XS platform, with the latter system based on Field Programmable Gate Array (FPGA) technology. For Fortum Power and Heat, VTT has performed independent, third-party verification of application functions related to, e.g., reactor protection, in the NPP I&C renewal projects LARA (Pakonen et al. 2014) and ELSA. ELSA systems are based on Rolls-Royce's Spinline platform. Finally, for Fennovoima, the future operator of the Hanhikivi NPP, VTT has evaluated the functional architecture of I&C systems.

3.2 *I&C application logic modeling*

VTT has developed a graphical tool (Pakonen et al. 2013) called MODCHK for verifying I&C application logic[1] designs, based on the NuSMV model checker.

[1]While the tool is intended for modeling function block diagrams, we discuss the verification of application *logic* rather than *software*, because we have also worked with FPGA logic, which is not software.

The tool is based on a manually constructed library of basic (elementary) function block elements. Manual modeling is necessary, since nuclear system vendors often use vendor-specific function blocks, and do not provide the source code implementation, only functional descriptions (Pakonen et al. 2013).

An example of an elementary function block specified in NuSMV's input language is shown in Listing 1. The block – a rising edge trigger – has one input $IN1$ and one output OUT.

In our tool, the input $IN1$ is represented using three signals: 1) the actual (in this case, binary) value, $IN1$, 2) a binary signal indicating signal *validity*, $IN1\_fault$, and 3) a binary signal indicating whether a wire is connected to the input in the block diagram, $IN1\_connected$.

```
MODULE R_TRIG(IN1, IN1_fault, IN1_connected)
VAR
  prevIN1 : boolean;
DEFINE
  OUT := !prevIN1 & IN1 & !IN1_fault;
  OUT_fault := IN1_fault;
ASSIGN
  init(prevIN1) := FALSE;
  next(prevIN1) :=
  case
    IN1_fault : prevIN1;
    !IN1_fault : IN1;
  esac;
```

Listing 1: NuSMV code for a rising edge trigger element

First, the actual value can be of binary (boolean) or analogue (integer) type.

Second, the "fault" signal indicates whether the signal status is set to valid ("OK" or "no fault") or invalid ("not OK" or "fault"), based on, e.g., a measurement failure, or loss of communication. In the application logic, the validity can then be taken into account in subsequent processing (Rolls-Royce 2012), for example by excluding data from a failed subsystem out of a majority voting logic (e.g. 2-out-of-4) (Areva NP 2008). (In Listing 1, the output is only set (e.g., a rising edge is detected) if the last *valid* input value was false and the current *valid* input value is true. The validity of the input is passed directly to the validity of output.) Validity processing has been a relevant factor in about 12% of the design issues that we have identified (See Table 1).

Finally, the "connected" signal can be used to specify how unconnected inputs are processed, without the user having to take such issues into account when drawing the diagram (in Listing 1, this feature is not used).

The blocks can also have parameters, in which case there is only a reference to the value. The associated graphical element can be specified using SVG (Scalable Vector Graphics).

It is also possible to define *composite function blocks*, and specify the internal logic by building a diagram out of other (elementary *or* composite) function blocks, enabling a multilevel hierarchy (Pakonen et al. 2013). Composite blocks are especially useful for modeling large applications that consist of several redundant subsystems (or "channels") replicating the same logic.

3.3 *I&C application logic verification*

After the model of the application logic has been constructed, verifiable properties are then formalized. In our tool, the formal properties are written using a simple text editor. We are in the process of collecting formal properties from our practical projects, in order to create a user-friendly specification tool that is suited to the I&C domain (Pakonen et al. 2016).

Based on the model and the properties, MODCHK will then create the corresponding input files for NuSMV. The basic idea of how the NuSMV model is generated can be seen by comparing the code in Listing 2 with Listing 1 and the function block diagram in Figure 2. (Since all the block inputs in Figure 2 have a connection, each "connected" signal has the value true.)

```
VAR
  a, a_fault, b, b_fault: boolean;
  NOT1 : NOT(a, a_fault, TRUE);
  OR1 : OR(NOT1.OUT, NOT1.OUT_FAULT, TRUE,
           b, b_fault, TRUE);
  R_TRIG1 : R_TRIG(OR1.OUT, OR1.OUT_fault, TRUE);
  RS1 : (a, a_fault, TRUE,
         R_TRIG1.OUT, R_TRIG1.OUT_fault, TRUE);
DEFINE
  c := RS1.OUT;
  c_fault := RS1.OUT_fault;
```

Listing 2: NuSMV code for the function block diagram in Figure 2

If NuSMV responds that a property is false, the counterexample is then transformed by MODCHK to a 2D animation that the user can replay back and forth. Colors, line styles, and numerical monitors are used to visualize the values of model variables at each time step. As an example, in Figure 1, there are four less-than blocks, and a 2-out-of-4 voting block. Since the first analogue signal has the value 5, the output of the first less-than block is true (thick, red wire). The second analogue signal has the value 0, also below the limit, but this signal is invalid (dashed wire). There are two true votes received by the voting block, but the invalid signal is not counted, so the output remains false (thin, black wire).

There are certain limitations (Pakonen et al. 2014) due to both our modeling approach and the underlying model checker NuSMV. Since the model is, in essence, a fairly straightforward FSM, complex control algorithms (e.g., PID) cannot be analyzed. Both the modeling of time (Lahtinen et al. 2012) and the discretization of analogue data are of particular concern, since the analyst must often resort to ad hoc "tweaking" in order to avoid state space explosion. Thankfully, direct errors made in either the model or in specifying the properties are very often revealed

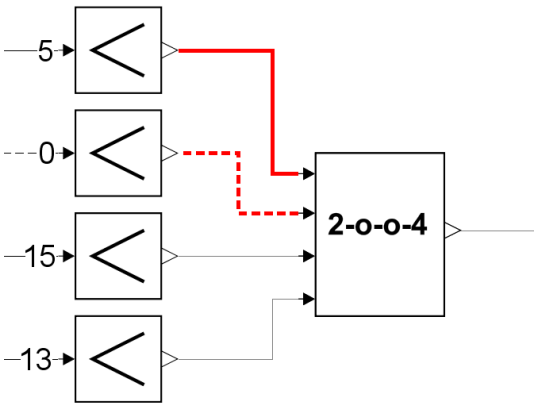

Figure 1: Partial screen capture showing counterexample visualization in the MODCHK tool

through the resulting counterexamples (Pakonen et al. 2014). In general, our approach is validated by successful practical experience, as demonstrated by the findings presented in Section 5.

3.4 *Related work*

A lot of the research on I&C model checking has focused on the automatic generation of the model based on standard programmable logic controller (PLC) languages – especially the IEC 61131-3 (e.g., (Ovatman et al. 2016), (Pavlovic and Ehrich 2010), (Yoo et al. 2009), (Adiego et al. 2015)) – or, in the case of FPGAs, directly based on hardware description languages like VHDL (Déharbe et al. 1998).

Another key research topic is user-friendly property specification. A popular collection of property specification patterns was published by Dwyer et al. (1999). Domain specific patterns have been suggested by, e.g., Campos and Machado (2009). Different graphical specification languages are listed in (Autili et al. 2007) and (Pakonen et al. 2016). Ljungkrantz et al. (2014) have collected formal industry requirements in order to develop a specification language suited for control engineering.

4 FORMAL SAFETY PROPERTIES

Requirement specification documents do not typically contain statements such as "there shall never be a spurious actuation". Such statements might be considered self-evident, or omitted due to the difficulty in their verification. In any case, an analyst performing formal verification must take specific care in addressing spurious actuation. For many properties that capture a desired ("good") behavior, there exist one or more complimentary properties that capture unintended ("bad") behavior.

Let us consider a common type of property: "a $request$ shall lead to a $response$", which can be written in LTL as:

$$\mathbf{G}(request \rightarrow response) \tag{1}$$

Note that (1) is actually *safety* property. It expresses a "good thing", but the $response$ is required immediately rather than eventually, and a finite counterexample is sufficient for demonstrating that the property does not hold. We can also rewrite (1) using the logically equivalent form: $\mathbf{G}\neg(request \wedge \neg response)$. The counterexample would obviously contain a "bad" state where the $request$ holds but the $response$ is not true.

However, (1) holds in a scenario where $response$ is true but $request$ is not. In order to address spurious actuation, let us turn (1) around to state: "a $response$ implies that there is a $request$", or:

$$\mathbf{G}(response \rightarrow request) \tag{2}$$

The equivalent form is now $\mathbf{G}\neg(response \wedge \neg request)$: "there shall never be a scenario where $response$ holds but $request$ is not true". The counterexample will then accordingly reveal a scenario leading to a spurious actuation.

In (1), we expect that the reaction is immediate, but there is often a delay before the $response$ is expected. In that case, we can formulate a *liveness* property: "a $request$ shall eventually lead to a $response$", or:

$$\mathbf{G}(request \rightarrow \mathbf{F}\ response) \tag{3}$$

In order to address spurious actuation, the counterpart safety property for (3) is:

$$\mathbf{G}(response \rightarrow \mathbf{O}\ request) \tag{4}$$

If past temporal operators cannot be used, (4) can be rewritten using the far less intuitive but equivalent (Pradella et al. 2003) expression $\neg(\neg request\ \mathbf{U}\ (response \wedge \neg request))$, or based on the "Precedence" (Dwyer et al. 1999) pattern: $(\mathbf{G}\neg response) \vee (\neg response\ \mathbf{U}\ request)$.

If it is known exactly after how many time steps $response$ should be true, the "good" property can be stated using (nested) **X** operators, e.g., for two time steps:

$$\mathbf{G}(request \rightarrow \mathbf{X}(\mathbf{X}\ response)) \tag{5}$$

The spurious actuation property that is complimentary to (5) can be stated using (nested) **Y** operators. Again, for two time steps:

$$\mathbf{G}(response \rightarrow \mathbf{Y}(\mathbf{Y}\ request)) \tag{6}$$

The formulas above exemplify why, as stated in (Benedetti and Cimatti 2003), **Y** and **O** are the "temporal duals" of **X** and **F**, respectively.

Note also that if (2) or (6) is true, (4) is also true. Likewise: if (1) or (5) is true, (3) is also true.

Out of the 1079 formal properties we have collected from our practical projects (Pakonen et al. 2016), 2.9% are of the type (3), and 1.9% are of the type (4). Since (1) and (2) have the same logical construct, we cannot distinguish between them, but their joint share is at 59.9% of all properties.

5 PRACTICAL EVALUATION

5.1 *List of identified design issues*

We have collected data on the design issues we have identified using model checking in practical customer projects for several customers between the years 2008 and 2016. A design issue, in this context, means that the analysis revealed a practical example of a potential scenario, where a certain chain of events leads to the I&C system application logic ending up in a state that was contrary to a stated functional requirement.

The data is presented in Table 1. In addition to a short, generalized description, for each design issue, we state whether the issue had certain characteristics (e.g., whether it was a spurious actuation scenario), as well as whether the logic that was analyzed contained some design elements that typically introduce complexity. (The ordering of the issues in Table 1 is based on these characteristics, rather than, e.g., the time of discovery.)

The reader should note that the issues are about a single (sub)system not acting according to its stated requirement in some scenario, however unlikely that scenario is. We wish to emphasize that: 1) we do not discuss the actual safety relevance of the issues, which, for any issue, may be insignificant and/or purely theoretical, 2) each system has been considered in isolation – without accounting for, e.g., the characteristics of the controlled process – and any issue might therefore be practically irrelevant in the actual context of these systems.

5.2 *A practical example*

As an example, let us consider one of the identified design issues that resulted in spurious actuation (namely, issue 4 in Table 1).

The verified application logic consisted of tens of function blocks, out of which have selected the ones that are needed for reproducing the scenario. An abstracted and modified diagram composed of these blocks can be found in Figure 2.

In this modified logic, we have two inputs, a and b, one output c, and four function blocks: NOT, OR, a rising edge trigger (see also Listing 1), and a Set-Reset flip-flop memory (with priority on the reset side).

Exemplar functional requirements for the logic include r_1: "If b is false and a is true, c shall be true", r_2: "If b changes to true, c shall be false", and r_3: "If a changes to false, c shall be false". Formalizing to LTL (omitting, for simplicity, the validity processing), we get p_1: $\mathbf{G}((\neg b \wedge a) \rightarrow c)$, p_2: $\mathbf{G}(\neg b \wedge \mathbf{X}b \rightarrow \mathbf{X}\neg c)$, and p_3: $\mathbf{G}(a \wedge \mathbf{X}\neg a \rightarrow \mathbf{X}\neg c)$. Furthermore, we can specify that spurious actuation should not occur, by stating p_s: $\mathbf{G}(c \rightarrow (\neg b \wedge a))$.

Verification results indicate that p_1 and p_2 are true, but p_3 and p_s are false. The model checker returns a

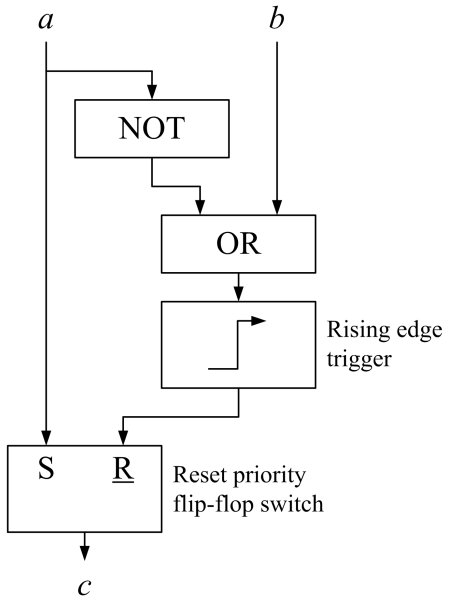

Figure 2: Example logic

(finite) counterexample, depicted in Figure 3. From the counterexample, we can see that if the falling edges of both a and b signals occur at the *exact* same time (processing cycle), then the output of the OR block is never false (since $(a \wedge \neg b)$ is never true in this scenario). Therefore, no rising edge will ever occur to reset the c signal. Since c remains true while a is false, spurious actuation occurs.

It would be challenging – although not impossible – to identify the issue using testing or simulation, because 1) the scenario involves two separate events occurring at the *exact* same time, and 2) b being true while c is false is a counter-intuitive circumstance in the actual context (as the "purpose" of b is to *reset* c).

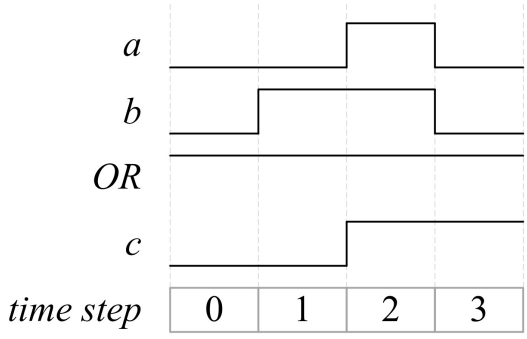

Figure 3: Counterexample for properties p_3 and p_s

5.3 *Analysis of the data*

The obvious conclusion based on the data in Table 1 is that model checking is an effective verification method for analyzing spurious actuation of I&C systems, by revealing design issues that might cause the application logic to inadvertently actuate a function. 37% of the 40 design issues we have detected were related to spurious actuation scenarios.

Almost all (with the exception of just two) identified issues depended upon at least one of the follow-

Table 1: List of generalized design issues identified with model checking.

	Issue characteristics					Elements in the design			
Generalized description	spu.	fro.	sev.	tim.	ope.	val.	mem.	del.	fb.
1. Short signal pulses interfere with a test logic and lead to actuation.	❍		❍	❍	❍		❍	❍	
2. Test signal stored in delay block leads to spurious actuation after test.	❍		❍				❍	❍	
3. Test signal stored in memory leads to spurious actuation after test.	❍						❍	❍	
4. A reset signal is ignored, if it arrives at a specific time.	❍			❍	❍		❍		
5. Due to maintenance activity, a signal remains set without a cause.	❍		❍		❍	❍	❍		
6. Due to maintenance activity, a signal is spuriously set.	❍		❍		❍		❍		
7. Invalid data on start up, improper operator action lead to actuation.	❍			❍	❍	❍	❍		❍
8. Spurious actuation commands are given on system start up.	❍							❍	
9. Two consecutive commands lead to an overtly long actuation signal.	❍			❍				❍	
10. Overlapping commands lead to an overtly long actuation signal.	❍			❍				❍	
11. Conflicting actuation commands are sent.	❍			❍				❍	
12. Invalid signals trigger conflicting commands.	❍					❍			
13. Conflicting inputs trigger conflicting commands.	❍								
14. A functional requirement is incorrect.	❍								
15. A safety command is given without valid actuation criteria.	❍			❍		❍			
16. Uncharacteristic inputs lead to inhibition of safety commands.		❍					❍	❍	❍
17. Configuration of delays means that a signal is blocked.		❍						❍	
18. Configuration of delays means that a signal is blocked.		❍	❍					❍	
19. After fluctuating inputs, operator can perform a forbidden action.				❍	❍		❍	❍	
20. Fluctuating input data leads to the incorrect operational state.				❍	❍		❍	❍	
21. Conflicting internal variables are set on fluctuating input data.				❍			❍	❍	
22. Redundant systems can be inhibited at the same time.			❍	❍			❍	❍	❍
23. A lower priority signal overrules a higher priority one.							❍		
24. Due to maintenance activity, a safety function is inhibited.			❍	❍	❍		❍		
25. Due to maintenance activity, an actuator is inhibited.			❍	❍	❍		❍		
26. A lower priority signal overrules a higher priority one.							❍		
27. If input signals are on during start up, a safety function is inhibited.							❍		
28. Redundant systems can be inhibited at the same time.				❍	❍		❍		❍
29. A test mode can only be deactivated via a wrong mechanism.					❍		❍		❍
30. On conflicting input data, no operational state is selected.			❍	❍			❍		
31. Conflicting operational modes selected on fluctuating input data.				❍	❍		❍		❍
32. An alarm can be acknowledged before it is received.				❍	❍		❍		
33. At system start up, an actuation command is inhibited.								❍	
34. A functional requirement on timing is technically incorrect.								❍	
35. Fluctuating inputs lead to short actuation command bursts.								❍	
36. A safety command is inhibited on system start up.								❍	❍
37. Very rapidly fluctuating commands are sent to an actuator.								❍	
38. Very rapidly fluctuating commands are sent to an actuator.								❍	
39. A safety command is inhibited after a delay.								❍	
40. Signal validity processing fails.						❍			
%	37	7.3	17	42	32	12	54	51	17

spu. = Spurious actuation
fro. = The logic is permanently frozen to an unwanted state (requires system restart to resolve).
sev. = The interaction of several systems is needed to reproduce the scenario.
tim. = The scenario involves very exact timing of events.
ope. = The scenario involves human operator actions.
val. = Signal validity processing is relevant to the scenario.
mem. = Memory components (e.g., flip-flop switch blocks) are used in the logic.
del. = Delay components are used in the logic.
fb. = Feedback loops are used in the logic.

ing elements being used in the logic design:

1. A **memory element**, such as a flip-flop switch or latch. Notably, a block that is not necessarily a memory element by definition can in practice serve as one, if, e.g., its processing is based on the last valid value. Relevant in 54% of the findings.

2. A **delay element**, or timed element, which also serves as a type of memory. Delay elements are often used in I&C systems, due to, e.g., the dynamic characteristics of the processes being controlled (Pakonen et al. 2016). Relevant in 51% of the findings.

3. A **feedback loop**, which also introduces a delay element out of necessity, since the processing order the blocks in the loop needs to be made explicit. Relevant in 17% of the findings.

4. **Detailed validity processing** in element(s) was relevant in 12% of the findings.

35% of the issues depended on more than one of the above-mentioned elements being used in the same logic.

One of the strengths of function block diagrams as a programming language is that it is relatively easy to understand the "flow" of processing from the inputs to the outputs. Notably, elements such as memory or delay blocks or feedback loops interfere with this flow, making it more difficult to assess the logic.

Other recurring features of the scenarios were:

1. 42% of the issues involved **exact timing** of external events, e.g., independent events occurring on the same processor cycle. Even with discrete-time model checking, detailed timing can be analyzed.

2. 32% of the issues involved **human user actions**, typically personnel (operation or maintenance) doing something ill-advised and/or ill-timed.

3. 17% of the issues required the **interaction between several systems** for the problem to occur. Analyzing the systems in isolation would not have revealed the issue.

4. 7% of the scenarios resulted in the a process signal freezing permanently on some fixed state (requiring, e.g., system restart for recovery).

It would also be interesting to find out the frequency of detected issues per analyzed system, or in more detail: how common was it for a logic that used, e.g., a feedback loop, to contain a design issue? However, such analysis is hard, since it is difficult to compare between systems that might have different rationale for breaking the design down into applications, functions, sheets, etc.

It should also be noted that usually, the designs that were analyzed had already undergone a verification and validation (V&V) process using other, more conventional methods. Accordingly, (since spurious actuation scenarios are harder to analyze than "failure to actuate" (Authén et al. 2016),) it is likely that spurious scenarios are over-represented in our data, in the sense that other issues had more likely already been identified and corrected. I.e., the share of potential design issues related to spurious actuation is not necessarily as large as our data would suggest.

5.4 *Topics for further research*

In this paper, we have only considered spurious actuation caused by issues in application logic design. There is also the need for system-level analysis, covering failures of I&C hardware and equipment, as well as actions of operators. Probabilistic risk assessment (PRA) can be used to assess the plant level effects of spurious actuation from *any* given source, for example hot shorts caused by a power or I&C cable fire (Authén et al. 2016). On the other hand, PRA cannot be used to analyze application logic in detail. Work is therefore underway to create a safety assessment approach that integrates PRA with model checking (Lahtinen and Björkman 2016). An integrated approach could be used to, e.g., verify the fault-tolerance of a system.

Complexity measures have been proposed for identifying the parts of I&C software that warrant more detailed inspection. Simensen et al. (2009) suggest a method based on different white-box and black-box measures for individual function blocks, and a metric for assessing the interconnection complexity of block diagrams. Tyrväinen et al. (2016) suggest a simpler method based on the number of feedback loops and complex function blocks, connections between blocks on a signal path, etc. Complexity of function blocks in the latter method is based on model checking experience (Lahtinen et al. 2012), and the authors list similar elements as in Section 5.3. Our data could be used to validate and further develop such complexity measures, especially if the design elements behind issues we list in Table 1 were characterized in more detail.

6 CONCLUSIONS

The difficulty in analyzing spurious actuation is tied to the problem of achieving 100% test coverage. The strength of model checking is that exhaustive coverage provides proof that something "bad" can *not* happen, in addition to proving that something "good" will always happen. Our practical experience in the nuclear industry proves that issues in I&C application logic design that might lead to spurious actuation can be identified with model checking. In fact, to the best of our knowledge, it is the only truly effective method available.

Still, the analyst using model checking has to take specific care that spurious actuation is properly considered, since the requirement specification documents (serving as the starting point for property formalization) might not explicitly address *un*wanted functionality. For any desired ("good") property, there might be several complimentary properties that deal with unintended ("bad") behavior.

A key obstacle for the wider adoption of formal verification methods is the perceived cost. With the development of practical, user-friendly, domain-specific tools, that cost can be alleviated, and the obvious advantages of model checking become very hard to dismiss. The alternative might be an accident waiting to happen.

REFERENCES

Adiego, B. F., D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, & V. M. G. Suárez (2015). Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics 11*(6), 1400–1410.

Alpern, B. & F. B. Schneider (1985). Defining liveness. *Information processing letters 21*(4), 181–185.

Areva NP (2008). Instrumentation and Control, TELEPERM XS, System Overview. Technical sheet.

Authén, S., O. Bäckström, J. Holmberg, M. Porthin, & T. Tyrväinen (2016). NKS-361, Modelling of DIgital I&C, MODIG – Interim report 2015. NKS-R report.

Autili, M., P. Inverardi, & P. Pelliccione (2007). Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering 14*(3), 293–340.

Behrmann, G., A. David, & K. G. Larsen (2004). A tutorial on UPPAAL. Number 3185 in LNCS, pp. 200–236.

Bel V, BfS, CNSC, CSN, ISTec, ONR, SSM, & STUK (2015). Licensing of safety critical software for nuclear regulators, Common position of international nuclear regulators and authorised tehcnical support organisations, Revision 2015. Technical report.

Benedetti, M. & A. Cimatti (2003). Bounded model checking for past LTL. *LNCS 2619*, 18–33.

Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill, & L. J. Hwang (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98*, 142–170.

Campos, J. C. & J. Machado (2009). Pattern-based analysis of automated production systems. In *13th IFAC Symposium on Information Control Problems in Manufacturing*, Moscow, Russia, pp. 972–977.

Cimatti, A., E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, & A. Tacchella (2002). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, Volume 2404 of *LNCS*, Copenhagen, Denmark. Springer.

Clarke, E. M., A. Biere, R. Raimi, & Y. Zhu (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design 19*(1), 7–34.

Clarke, E. M., O. Grumberg, & D. Peled (1999). *Model checking*. MIT press.

Déharbe, D., S. Shankar, & E. M. Clarke (1998). Model checking VHDL with CV. *LNCS 1522*, 508–513.

Dwyer, M. B., G. S. Avrunin, & J. C. Corbett (1999). Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE 1999)*, pp. 411–420. IEEE.

IAEA (2015). *Technical Challenges in the Application and Licensing of Digital Instrumentation and Control Systems in Nuclear Power Plants*. Vienna: International Atomic Energy Agency.

Lahtinen, J. & K. Björkman (2016). Integrating model checking and PRA: a novel safety assessment approach for digital I&C systems. In *26th European Safety and Reliability (ESREL 2016)*, Glasgow, UK. CRC Press.

Lahtinen, J., J. Valkonen, K. Björkman, J. Frits, I. Niemelä, & K. Heljanko (2012). Model checking of safety-critical software in the nuclear engineering domain. *Reliability Engineering and System Safety 105*, 104–113.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions in Software Engineering SE-3*(2), 125–143.

Ljungkrantz, O., K. Åkesson, M. Fabian, & A. H. Ebrahimi (2014). An empirical study of control logic specifications for programmable logic controllers. *Empirical Software Engineering 19*(3), 655–677.

MDEP (2016). Multinational Design Evaluation Programme, DICWG-10, Common position on hazard identification and controls for digital instrumentation and control systems. Technical report.

Ovatman, T., A. Aral, D. Polat, & A. O. Ünver (2016). An overview of model checking practices on verification of plc software. *Software & Systems Modeling 15*(4), 937–960.

Pakonen, A., T. Mätäsniemi, J. Lahtinen, & T. Karhela (2013). A toolset for model checking of PLC software. In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA 2013)*, pp. 1–6. IEEE.

Pakonen, A., C. Pang, I. Buzhinsky, & V. Vyatkin (2016). Userfriendly formal specification languages – conclusions drawn from industrial experience on model checking. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*, pp. 1–8. IEEE.

Pakonen, A., J. Valkonen, S. Matinaho, & M. Hartikainen (2014). Model checking for licensing support in the Finnish nuclear industry. In *International Symposium on Future I&C for Nuclear Power Plants (ISOFIC 2014)*.

Pavlovic, O. & H. D. Ehrich (2010). Model checking PLC software written in function block diagram. In *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 439–448.

Pradella, M., P. San Pietro, P. Spoletini, & A. Morzenti (2003). Practical model checking of LTL with past. In *1st International Workshop on Automated Technology for Verification and Analysis (ATVA03)*.

Rolls-Royce (2012). Spinline™, A Rolls-Royce modular I&C digital platform dedicated to nuclear safety. Technical sheet.

Simensen, J. E., C. Gerst, B. A. Gran, J. Märtz, & H. Miedl (2009). Establishing the correlation between complexity and a reliability metric for software digital I&C-systems. In *28th International Conference on Computer Safety, Reliability, & Security (SAFECOMP 2009)*, Hamburg, Germany, pp. 55–66.

Tyrväinen, T., Bäckström, J.-E. Holmberg, & M. Porthin (2016). SICA - a Software Complexity Analysis Method for the Failure Probability Estimation. In *13th International Conference on Probabilistic Safety Assessment and Management, (PSAM 13)*, Seoul, Korea. International Association for Probabilistic Safety Assessment and Management, IAPSAM.

Yoo, J., S. Cha, & E. Jee (2009). Verification of PLC programs written in FBD with VIS. *Nuclear Engineering and Technology 41*(1), 79–90.